DUAL LEARNING OF CODE TRANSFORMATION PATTERNS FROM DEMONSTRATIONS WITH GENERATION-DISCRIMINATION OBJECTIVES

ABSTRACT

Large-scale software systems frequently require consistent code changes across multiple locations to address security vulnerabilities, improve performance, or add new features. While these changes share the same semantic intent, automating them is challenging due to syntactic variations in different code contexts. We present a novel deep learning approach that learns code transformations from examples and generalizes them across diverse contexts while preserving the original semantic intent. Our key contribution is a dual-learning adaptation technique that optimizes two objectives during model fine-tuning: generating accurate code transformations and discriminating between related and unrelated transformation patterns. On real-world code transformation tasks from the Linux Kernel, our approach achieves 20-21% higher Exact Match accuracy compared to standard supervised fine-tuning across different model sizes and demonstrates a 35% improvement over the traditional pattern mining method. These results show that our dual-learning strategy effectively can capture better transformation patterns from the demonstrations than the baselines to generate correct transformations from the input code.

1 INTRODUCTION

Software systems continuously evolve to address security vulnerabilities, enhance performance, fix bugs, and implement new features ((Padioleau et al., 2006)). This evolution often involves changes that must be consistently applied across multiple parts of the codebase. For instance, when an API is updated or deprecated, developers need to modify all code locations that use the old API to maintain consistency and prevent potential bugs (Lamothe et al. (2022); Li et al. (2013)).

In a large software system evolution, making repetitive code changes can pose significant challenges due to syntactic variations in semantically equivalent code. Figure (1) demonstrates this through the evolution of timer initialization in the Linux kernel. There are three changes (C1, C2, and C3) where multiple initialization steps of the timer are simplified into a single function call. Each change transforms initialization using `init_timer`, data assignment, and callback registration into a unified `setup_timer` call. While these changes share identical semantic intent, they exhibit syntactical variations in their implementation, such as varying variable names (e.g., `bwi_timer` in C1 versus `se_active_timer` in C2) and different callback functions (e.g., `st21nfca_se_wt_timeout` in C1 versus `sym53c8xx_timer` in C3). These variations cause traditional patch-based automation approaches ineffective. This migration spans nearly a decade: starting with 73 changes in 2008, continuing with sporadic updates and increased activities during 2014-2016 (43, 93, and 37 changes), before concluding with a large-scale removal of 267 `init_timer` calls in 2017 (Serrano et al. (2020)). This prolonged migration, spanning nearly a decade, serves as evidence of the inherent difficulty in performing such an evolution.

The repetitive patterns in these changes, despite their syntactic variations, present an opportunity for automation through deep learning. Deep learning models excel at recognizing underlying structures while remaining robust to surface-level differences (Yin et al., 2018), making them well-suited for automating code transformations that share semantic intent but vary in implementation.

In this work, we propose a deep learning-based approach to automate software evolution, where the model infer transformation patterns from multiple change examples at inference time and generates the transformed code for a given input. To further enhance the performance of deep learning in code transformation, we propose a dual-learning adaptation technique that simultaneously optimizes two objectives during fine-tuning: (1) a generation objective that predicts the next tokens for code trans-

```
// C1
- init_timer(&info->se_info.bwi_timer);
- info->se_info.bwi_timer.data = (unsigned long)info;
- info->se_info.bwi_timer.function = st21nfca_se_wt_timeout;
+ setup_timer(&info->se_info.bwi_timer, st21nfca_se_wt_timeout,
    (unsigned long)info);

// C2
- init_timer(&info->se_info.se_active_timer);
- info->se_info.se_active_timer.data = (unsigned long)info;
- info->se_info.se_active_timer.function =
    st21nfca_se_activation_timeout;
+ setup_timer(&info->se_info.se_active_timer,
    st21nfca_se_activation_timeout,
    (unsigned long)info);

// C3
- init_timer(&np->s.timer);
- np->s.timer.data = (unsigned long)np;
- np->s.timer.function = sym53c8xx_timer;
+ setup_timer(&np->s.timer, sym53c8xx_timer, (unsigned long)np);
```

Figure 1: Examples of timer initialization changes in the Linux kernel, showing the evolution from multiple initialization steps to a single `setup_timer` call. These changes are semantically equivalent, where the goal is to simplify the timer initialization.

formations, and (2) a relevance objective that learns to distinguish between related and unrelated transformation patterns through discrimination learning. This dual objective enables the model to both generate accurate transformations and recognize semantic relationships between different transformation instances. Through this approach, our model can effectively adapt transformation patterns from demonstrations to new inputs with different syntactic variations.

We evaluate our dual-learning adaptation technique on real-world code transformation tasks from the Linux Kernel. The results demonstrate that our approach outperforms the standard supervised fine-tuning by 20-21% in Exact Match accuracy across various model sizes. When compared to the state-of-the-art pattern mining method for the Linux Kernel code transformation (Serrano et al., 2020), our technique shows a substantial improvement of 35% in Exact Match, highlighting the effectiveness of our dual-learning strategy to learn code transformation patterns from demonstrations and apply it to the new input code. The replication package is available at `https://anonymous.4open.science/r/c-tuning/`.

2 Related Works

Deep learning for code transformations. Deep learning models have demonstrated strong performance across various software engineering tasks. Our work shares similarities with automated program translation (Yang et al., 2024; Pan et al., 2024; Li et al., 2024; Liu et al., 2023), which transforms programs between different programming languages while preserving semantics. While program translation operates across languages, our work focuses on evolution-driven transformations within the same language. Our approach also relates to automated code repair (Zhang et al., 2024; Fan et al., 2023; Hossain et al., 2024; Fu, 2023), where models learn to identify and fix buggy code patterns. Both code repair and our work involve learning transformation patterns to maintain system correctness, though we focus specifically on systematic evolution changes rather than individual bug fixes. While the approach like Dilhara et al. (2024) focus on addressing limited change examples in and Tufano et al. (2019) explore sequence-to-sequence models for code change automation, our work introduces a novel dual learning objective that better captures both the generation of transformed code and the relevance between different transformation patterns.

Pattern-based code transformation. Several approaches have been proposed to learn code transformations from examples. Early works like Sydit (Meng et al., 2011) and GENPAT (Jiang et al.,

2019) focused on inferring transformations from single change examples, limiting their applicability to complex evolution scenarios. LASE (Meng et al., 2013) advanced this by developing Abstract Syntax Tree (AST) edit sequences from multiple examples, though it struggles with changes incorporating multiple variants. REFAZER (Rolim et al., 2017) introduced a domain-specific language for transformation rules and employs clustering to handle variants, but like LASE, does not consider control-flow dependencies. PyEvolve (Dilhara et al., 2023) employs graph-based matching and specialized adaptation techniques but is specifically tailored for Python. Transformations in the Linux kernel present unique challenges due to complex changes with identical semantics but diverse syntax variants. Among pattern-based approaches, Spinfer (Serrano et al., 2020) demonstrates superior performance by considering both control-flow dependencies and multiple change variants when deriving semantic patches for the Linux kernel. Given its effectiveness compared to other pattern-based approaches, we use Spinfer as our baseline. In contrast to these pattern-based methods, our approach leverages deep learning to naturally handle syntactic variations while capturing semantic patterns, making it more adaptable across different contexts.

Dual Learning Framework. Prior works have explored dual learning frameworks to enhance code generation tasks (Wei et al., 2019; Wang et al., 2024; Ye et al., 2020). These approaches primarily focus on natural language to code generation tasks, using code summarization (code to natural language) as an auxiliary objective. Our work differs fundamentally as we apply dual learning to code transformation tasks, where we combine generative learning with a discrimination objective. This novel combination enables our model to both generate accurate transformations and learn the relationships between different transformation patterns, leading to better generation performance. Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) similarly leverage dual learning through generator and discriminator models to improve generation quality. However, unlike GANs that rely on separate networks trained adversarially, our approach utilizes a single model that jointly optimizes both generation and discrimination objectives. Furthermore, while GANs focus on distribution matching between real and generated samples, our discrimination objective specifically targets the semantic relationships between transformation patterns from code change examples.

3 METHODOLOGY

3.1 TASK DEFINITION

Given a set of demonstrations $\mathcal{D} = \{(x_1, y_1), ..., (x_k, y_k)\}$ where each pair consists of input code x_i and its transformed version y_i, and a new input code x_{new}, the goal is to generate the target transformed code y_{new}. Each demonstration pair captures the same underlying transformation pattern that needs to be applied to x_{new}, as illustrated in Figure 1. Formally, we aim to model the conditional probability in Equation 1,

$$p(y_{\text{new}}|x_{\text{new}}, \mathcal{D}; \theta) \tag{1}$$

where θ represents the parameters of our neural model.

During training, we have a dataset of instances $\mathcal{T} = \{(\mathcal{D}_i, x^i_{\text{new}}, y^i_{\text{new}})\}_{i=1}^N$ where each instance contains a set of demonstrations, an input code, and its target transformation. The standard supervised fine-tuning objective maximizes the log-likelihood as in Equation 2.

$$\mathcal{L}_{\text{sup}} = \sum_{i=1}^{N} \log p(y^i_{\text{new}}|x^i_{\text{new}}, \mathcal{D}_i; \theta) \tag{2}$$

At inference time, given a new set of demonstrations $\mathcal{D}_{\text{test}}$ and input code x_{test}, the model generates the transformed code by Equation 3.

$$y_{\text{test}} = \arg\max_y p(y|x_{\text{test}}, \mathcal{D}_{\text{test}}; \theta) \tag{3}$$

3.2 FINE-TUNING WITH DUAL OBJECTIVES FOR CODE TRANSFORMATION LEARNING

To better capture the semantic relationships between code transformations, we propose a dual learning paradigm that combines generation and discrimination objectives. In the fine-tuning stage, one training instance is a triplet of $(\mathcal{T}_a, \mathcal{T}_p, \mathcal{T}_n)$ where $\mathcal{T}_a$ serves as an anchor, $\mathcal{T}_p$ as a positive example sharing the same transformation semantics as the anchor, and $\mathcal{T}_n$ as a negative example that have different transformation semantics as $\mathcal{T}_a$.

The total training objective consists of two components as shown in Equation 4

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{gen}} + \lambda \mathcal{L}_{\text{dis}} \tag{4}$$

The generation loss $\mathcal{L}_{\text{gen}}$ is the standard next-token prediction objective computed for all instances in the triplet as shown by Equation 5.

$$\mathcal{L}_{\text{gen}} = \sum_{t \in \{a,p,n\}} \log p(y^t_{\text{new}} | x^t_{\text{new}}, \mathcal{D}_t; \theta) \tag{5}$$

The discrimination loss $\mathcal{L}_{\text{dis}}$ is a binary classification objective that encourages the model to distinguish between related and unrelated transformation patterns. To compute this loss, we first obtain the representation of each instance ($\mathcal{T}_a$, $\mathcal{T}_p$, $\mathcal{T}_n$) from the decoder's final layer hidden states. Specifically, we use the last token's hidden state vector which can attend to all previous tokens in the sequence. Let h_a^{last}, h_p^{last}, and h_n^{last} represent the last token's hidden states for the anchor, positive, and negative instances respectively. The discrimination loss is then computed as in the Equation 6,

$$\mathcal{L}_{\text{dis}} = -\log p(1|[h_a^{\text{last}}; h_p^{\text{last}}]) - \log p(0|[h_a^{\text{last}}; h_n^{\text{last}}]) \tag{6}$$

where $[;]$ denotes vector concatenation. The model learns to assign high probability when two instances share the same transformation semantics (anchor and positive) and low probability when they have different semantics (anchor and negative). Through this discrimination objective, the model develops the ability to recognize and distinguish different transformation patterns.

By utilizing all instances in the triplet for the generative loss $\mathcal{L}_{\text{gen}}$, our approach maximizes the learning signal from each training batch. This design allows the model to learn from multiple transformation instances simultaneously while also learning to distinguish between semantically similar and different transformations through $\mathcal{L}_{\text{dis}}$. The dual learning paradigm thus enhances the model's ability to both generate accurate transformations and understand the underlying semantic relationships between different transformation patterns.

4 EXPERIMENTAL SETTING

4.1 DATASET

Our dataset is derived from real-world C source code changes in the Linux kernel codebase, originally collected in Serrano et al. (2020). Each instance consists of a pair of C code snippets representing the state before and after a change. These changes are organized into directories, where each directory contains changes that share the same semantic intent while having different syntactic forms, as illustrated in Figure 1. For each function in the code, we extract the modified lines before and after the transformation, which become the input and target code respectively.

We construct training triplets of an anchor, a positive example, and a negative example $(\mathcal{T}_a, \mathcal{T}_p, \mathcal{T}_n)$. Each $\mathcal{T}$ represents a transformation instance defined as: $\mathcal{T} = (\mathcal{D}, x, y)$ Here, $\mathcal{D} = \{(x_1, y_1), ..., (x_k, y_k)\}$ is a set of demonstrations, x is the input code to be transformed, and y is its target transformation. For each transformation instance, we randomly sample up to N_d demonstrations to create $\mathcal{D}$. To ensure diversity in the training data, we perform N_s rounds of sampling for each triplet. We set both N_d and N_s to 5, striking a balance between capturing sufficient variations and maintaining computational feasibility as we fine-tune the model in our local machine. For the testing set, we use directories outside those that are used to create the training triplets. The final dataset contains 19,143 triplets for training and 632 instances for testing.

Model Size	Setting	EM	CodeBLEU	BLEU	METEOR	ROUGE	chrF
135M	NTP	0.18	0.51	0.62	0.71	0.62	63.11
	NTP + CLS (Ours)	**0.39**	**0.62**	**0.75**	**0.82**	**0.74**	**76.95**
360M	NTP	0.58	0.72	0.84	0.87	0.84	87.03
	NTP + CLS (Ours)	**0.78**	**0.81**	**0.91**	**0.93**	**0.91**	**92.61**
-	Spinfer	0.43	0.44	0.50	0.51	0.50	51.17

Table 1: Performance comparison between our dual-learning framework and standard next token prediction (NTP). All metrics show consistent improvements over the baseline NTP and Spinfer.

4.2 Models, baselines, and metrics

Models. We leverage SmolLM2 (Allal et al. (2025)) as our base model. We experiment with two size variants: 135M[1] and 360M[2] parameters. We choose these models as these are one the latest model that can still be fine-tuned in our local machine.

Baselines. We evaluate our dual-objective fine-tuning approach against several baselines:

- **Next Token Prediction (NTP).** The base model trained solely with the standard language modeling objective (Equation 5). This baseline demonstrates the impact of incorporating the discrimination loss in our approach.
- **Spinfer.** The current state-of-the-art approach for Linux Kernel code transformation (Serrano et al. (2020)). Unlike our learning-based method, Spinfer employs clustering and pattern mining techniques to analyze demonstrations and infer transformation rules.
- **Contrastive Learning.** We compare against two established contrastive learning approaches: InfoNCE (van den Oord et al. (2018)) and Triplet Loss (Schroff et al. (2015)). Both methods learn semantic similarities between anchors, positive examples, and negative examples in the embedding space. We also explore whether incorporating these contrastive objectives into our method yields additional improvements.

Metrics. We employ multiple automated metrics that capture different aspects of code generation quality. We use Exact Match (EM) to measure perfect reproduction of the target code, and CodeBLEU (Ren et al. (2020)) which specifically evaluates code similarity by considering syntactic and semantic features. For general sequence similarity, we utilize smoothed BLEU-4 (Papineni et al. (2002)), which measures precision by counting matching sequences between generated and reference code, and METEOR (Banerjee & Lavie (2005)), which flexibly matches similar tokens and considers word order, both widely adopted in generation tasks (Yusuf et al. (2023); Roy et al. (2021)). We complement these with ROUGE@2 (Lin (2004)), which focuses on recall by checking how much of the reference code is captured in the generation, and chrF (character n-gram order=6) (Popovic (2015)) to capture character-level similarities, which is particularly relevant for code where small differences can be significant.

5 Results

5.1 Comparison with next token prediction objective

Table 1 presents the comparative evaluation of our dual-learning framework against the baseline next token prediction (NTP) approach across two model sizes: 135M and 360M parameters. Our approach (NTP + CLS) demonstrates consistent improvements across all metrics for both model sizes. For the 135M model, adding the discrimination objective improves the exact match by 21% (from 0.18 to 0.39) and achieves higher scores across all other metrics, with improvements ranging

[1] https://huggingface.co/HuggingFaceTB/SmolLM2-135M
[2] https://huggingface.co/HuggingFaceTB/SmolLM2-360M

Model Size	Setting	EM	CodeBLEU	BLEU	METEOR	ROUGE	chrF
135M	NTP + InfoNCE	0.18	0.52	0.62	0.72	0.62	63.00
	NTP + Triplet	0.18	0.51	0.62	0.71	0.62	62.76
	NTP + CLS + InfoNCE	0.38	0.62	0.74	0.81	0.73	76.30
	NTP + CLS + Triplet	**0.41**	**0.63**	**0.75**	**0.82**	**0.74**	**77.05**
	NTP + CLS (Ours)	0.39	0.62	**0.75**	**0.82**	0.74	76.95
360M	NTP + InfoNCE	0.57	0.72	0.83	0.86	0.83	86.09
	NTP + Triplet	0.57	0.72	0.83	0.87	0.83	86.09
	NTP + CLS + InfoNCE	0.77	**0.81**	**0.91**	**0.93**	**0.91**	**92.72**
	NTP + CLS + Triplet	**0.78**	0.80	0.90	0.92	0.91	92.34
	NTP + CLS (Ours)	**0.78**	**0.81**	**0.91**	**0.93**	**0.91**	92.61

Table 2: Performance comparison of our dual-learning framework (NTP + CLS) against contrastive learning objectives (InfoNCE and Triplet Loss). Our approach consistently outperforms other methods across all metrics, demonstrating its effectiveness in enhancing model performance.

from 11-22%. The improvement of our approach is also similar with the 360M model, where we observe gains in exact match by 20% (from 0.58 to 0.78) and consistent improvements across all other metrics, with gains ranging from 9-13% in the remaining metrics. These results demonstrate that incorporating a discrimination objective alongside next token prediction leads to more robust and accurate code transformation, with the benefits becoming more pronounced as model size increases.

5.2 Comparison with Spinfer

Our NTP + CLS approach with the 360M parameter model demonstrates substantial improvements over the traditional pattern-mining Spinfer (Table 1). The 360M parameter model with NTP + CLS achieves an exact match score of 0.78, indicating a 35% improvement over Spinfer (0.43). This significant performance gap is consistent across all evaluation metrics: CodeBLEU (0.81 vs 0.44, +37%), BLEU (0.91 vs 0.50, +41%), METEOR (0.93 vs 0.51, +42%), ROUGE (0.91 vs 0.50, +41%), and chrF (92.61 vs 51.17, +41.44%). Notably, even our baseline NTP in the 360M model substantially outperforms Spinfer, achieving a 15% higher exact match (0.58 vs 0.43) with consistent improvements across all metrics.

Our 135M parameter model reveals interesting trade-offs in the context of code transformation. When compared to Spinfer, our NTP + CLS model exhibits a distinct performance pattern: while achieving a lower exact match score (0.39 vs 0.43), it consistently outperforms Spinfer across similarity-based metrics, namely: CodeBLEU (0.62 vs 0.44), BLEU (0.75 vs 0.50), METEOR (0.82 vs 0.51), ROUGE (0.74 vs 0.50), and chrF (76.95 vs 51.17). This pattern can be attributed to the fundamental differences in failure handling between the two approaches. Spinfer produces no output when it fails to find a matching pattern, leading to lower scores in similarity-based metrics despite maintaining a reasonable exact match. In contrast, our neural model always attempts to generate a transformation. While these outputs may not exactly match the ground truth, they often capture partial correctness, resulting in higher similarity scores. This behavior suggests that while our 135M model demonstrates better generalization capabilities than Spinfer, it requires additional capacity to consistently produce exact matches. These findings highlight the importance of model size in achieving good performance across all metrics in the code transformation task.

5.3 Comparing discrimination and contrastive learning objectives

To better understand the role of discrimination objective in learning relationships between positive and negative change examples, we conducted two sets of experiments. First, we examine whether discrimination is more effective than alternative approaches by replacing the discrimination objective in our dual learning framework with two contrastive learning objectives: InfoNCE (van den Oord et al. (2018)) and TripletLoss (Schroff et al. (2015)). Then, we investigate if combining these objectives with discrimination could yield better performance. Both InfoNCE and TripletLoss are designed to learn relationships between examples, but through different mechanisms. TripletLoss

Model Size	Setting	EM	CodeBLEU	BLEU	METEOR	ROUGE	chrF
135M	Mean Pooling	0.15	0.50	0.60	0.70	0.60	61.19
	Max Pooling	0.08	0.42	0.49	0.58	0.49	53.06
	Last Token	**0.39**	**0.62**	**0.75**	**0.82**	**0.74**	**76.95**
360M	Mean Pooling	0.59	0.72	0.84	0.87	0.83	86.39
	Max Pooling	0.56	0.71	0.82	0.86	0.82	85.50
	Last Token	**0.78**	**0.81**	**0.91**	**0.93**	**0.91**	**92.61**

Table 3: Performance comparison of mean pooling and max pooling versus last token representation across different model sizes. The last token approach consistently outperforms mean pooling across all evaluation metrics.

enforces a fixed margin between similarities, ensuring the distance between an input and its positive example is smaller than the distance to the negative examples by at least a specified margin. In contrast, InfoNCE learns through normalized similarity scores, maximizing the probability of identifying the positive example using a softmax over similarity scores between an anchor and all other examples (both positive and negative).

Table 2 shows that both NTP + InfoNCE and NTP + Triplet Loss perform worse compared to our NTP + CLS approach. For the 135M model, NTP + InfoNCE achieves an exact match of 0.18 and NTP + Triplet achieves 0.18, while NTP + CLS reaches 0.39, showing a substantial improvement of 21%. This performance gap is consistent across other metrics, with NTP + CLS outperforming both contrastive learning variants by 10-13% in CodeBLEU, BLEU, METEOR, ROUGE, and chrF. The gap widens further with the 360M model, where NTP + CLS achieves an exact match of 0.78, outperforming NTP + InfoNCE (0.57) and NTP + Triplet (0.57) by 21%. Similar improvements are observed across all metrics, with NTP + CLS showing 9-11% better performance. These results demonstrate that our discrimination-based approach is more effective than contrastive learning objectives for this task.

Table 2 further shows that incorporating additional contrastive learning objectives (InfoNCE or Triplet Loss) to our NTP + CLS framework yields slight improvements. For the 135M model, NTP + CLS + Triplet Loss achieves a higher exact match of 0.41 compared to 0.39 of NTP + CLS, while NTP + CLS + InfoNCE achieves 0.38. Other metrics show similar patterns with both variants showing 0-0.01 point improvements in BLEU (0.74-0.75 vs 0.75) and ROUGE (0.73-0.74 vs 0.74), and about 0.1 point difference in chrF score. For the 360M model, both NTP + CLS + InfoNCE and NTP + CLS + Triplet achieve comparable exact match scores (0.77-0.78) to NTP + CLS (0.78). Other metrics follow the same pattern with minimal differences across BLEU, METEOR, and ROUGE scores, and less than 0.1 points in chrF score. These results suggest that while adding contrastive objectives to NTP + CLS can provide marginal benefits, the discrimination objective alone achieves strong performance.

5.4 Impact of token representation strategies on generation performance

For the discrimination objective, we extract vector representations for comparing target pairs with positive and negative examples. Our implementation utilizes the last token's representation from the final decoder layer, leveraging its ability to attend to all prior tokens in the sequence. To validate this design choice, we conducted a comparative analysis between this approach and an alternative that uses mean pooling and max pooling for aggregating representations. The subsequent steps remain consistent across both variants: we concatenate the input pair's representation with that of the positive example and train the model to predict 1 for relevant pairs, while predicting 0 for negative (irrelevant) pairs.

The results in Table 3 demonstrate that the last token approach consistently outperforms both mean pooling and max pooling across all metrics. For the 135M model, using the last token representation significantly improves the exact match from 15% to 39%, while max pooling performs notably worse with only 8% exact match. This improvement pattern is consistent in the 360M model, where the last token approach achieves 78% exact match compared to 59% with mean pooling and 56% with

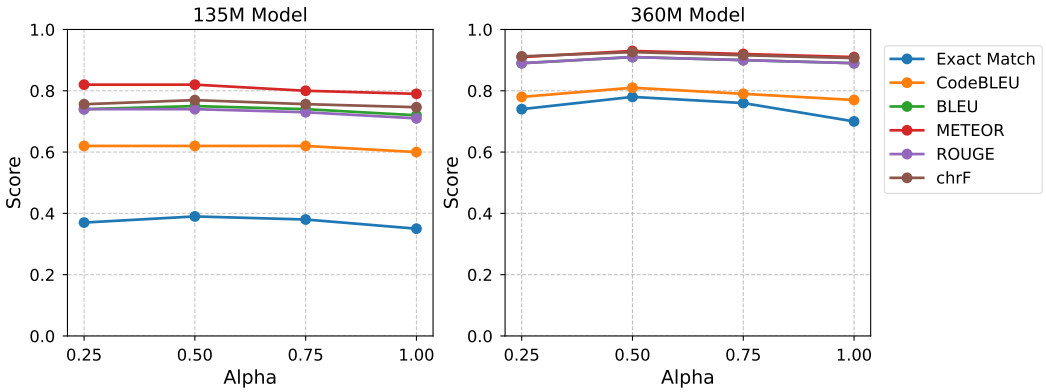

Figure 2: Performance comparison of different metrics across two model sizes (135M and 360M parameters) with varying α values. For both models, ($\alpha = 0.5$) generally yield better performance across all metrics. All metrics are normalized to the range [0,1] for consistent comparison.

max pooling. The improvements extend beyond exact match, with the last token approach showing relative gains of 12-15% across CodeBLEU, BLEU, METEOR, ROUGE, and chrF metrics for the 135M model, and 9-11% for the 360M model. These findings support our hypothesis that the last token's ability to attend to all prior tokens makes it a more effective representation strategy than both mean pooling and max pooling approaches.

5.5 Impact of different alpha values on generation performance

The alpha value (α) controls the contribution of discrimination loss in our dual learning objective (Equation 4), where higher values emphasize the model's ability to distinguish between different transformation patterns. To understand its impact better, we evaluated model performance across multiple alpha values: $\{0.25, 0.5, 0.75, 1.0\}$.

The results in Figure 2 show that the alpha (α) parameter plays a key role in model performance. Both models perform best with α=0.5, where the 135M model reaches 0.39 exact match score while the 360M model achieves 0.78. We found that the 360M model's performance drops notably when alpha deviates from 0.5, falling to 0.70 at α=1.0. The same pattern holds across other metrics where α=0.5 consistently yields the best results. The smaller 135M model follows a similar trend, where the performance peaks at α=0.5 with a score of 0.39, and gradually declines to 0.35 when α=1.0, though the variation is less pronounced compared to the 360M model. These results highlight an important practical finding: careful tuning of alpha is crucial for both model sizes. Setting α=0.5 yields the best result in the code transformation task.

5.6 Impact of training triplet count on generation performance

Understanding how our approach's performance scales with training data is crucial for real-world deployments, where data availability may vary. Figure 3 presents the relationship between the number of triplets seen during fine-tuning and exact match for two model architectures: 135M and 360M parameters. We compare three model variants: NTP + CLS (Ours), NTP, and Spinfer (shown as a red dashed line).

For the 135M model, increasing the number of training triplets leads to substantial improvements in exact match for both NTP + CLS and NTP. Our approach consistently outperforms the NTP baseline across all data regimes. With limited data (3,000 triplets), both approaches struggle to learn meaningful patterns, achieving 0% exact match. However, as we increase the training data to 9,000 triplets, NTP + CLS shows a significant jump to 24% exact match, compared to NTP's 11%. This improvement continues with the full dataset (19,140 triplets), where NTP + CLS achieves 39% exact match. However, this result still falling slightly short of Spinfer's 43% baseline. While this represents a significant improvement, it still falls short of Spinfer with 43% exact match, suggesting that the limited capacity of the 135M model may be a bottleneck.

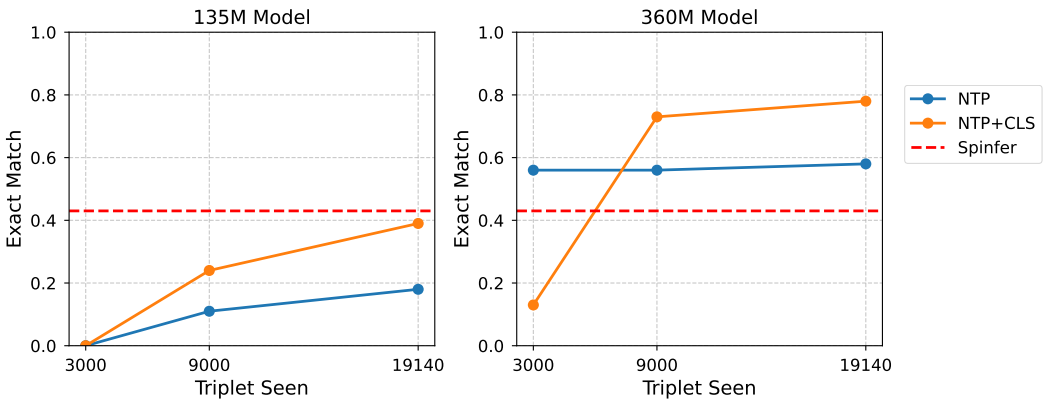

Figure 3: Comparison of exact match accuracy between NTP + CLS and NTP across different training data sizes. The exact match of NTP + CLS shows steady improvement with increasing data until approximately 70% of the training data, after which the gains become more gradual.

For 360M model, NTP + CLS exhibits substantial gains as more triplets are seen during fine-tuning. Starting from 13% exact match with 3,000 triplets, it rapidly improves to 73% with 9,000 triplets. The improvement continues with the full dataset, reaching 78% exact match, though the rate of gain starts to diminish after 9,000 triplets. This pattern suggests that while additional training data beyond 9,000 triplets still contributes to performance improvements, the marginal benefits begin to diminish. Interestingly, the baseline NTP demonstrates strong initial performance, achieving 56% exact match even with limited data. However, the performance does not increase further after seeing more data during fine-tuning. Both neural approaches (NTP + CLS and NTP) eventually outperform Spinfer's 43% baseline in this larger model setting. These results highlight the importance of model capacity in effectively learning code transformation patterns from the training data.

6 Conclusion

In this work, we presented a deep learning-based approach for automating consistent code transformations in large-scale software systems. By leveraging a dual-learning adaptation technique, our model simultaneously optimizes code generation and pattern discrimination, enabling it to generalize transformations across diverse contexts while preserving semantic intent. Our evaluation on real-world Linux Kernel transformation tasks demonstrated that our method outperforms standard supervised fine-tuning by 20-21% in Exact Match accuracy and achieves a 35% improvement over the traditional pattern mining technique. These results highlight the effectiveness of our approach in capturing and applying transformation patterns from demonstrations.

Future work includes extending our framework to handle more complex transformations involving structural code modifications, incorporating broader context, and exploring techniques to further enhance model interpretability and trustworthiness for the resulting transformation code.

References

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, et al. Smollm2: When smol goes big–data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*, 2025.

Satanjeev Banerjee and Alon Lavie. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In *IEEvaluation@ACL*, pp. 65–72. Association for Computational Linguistics, 2005.

Malinda Dilhara, Danny Dig, and Ameya Ketkar. PYEVOLVE: automating frequent code changes in python ML systems. In *ICSE*, pp. 995–1007. IEEE, 2023.

Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. Unprecedented code change automation: The fusion of llms and transformation by example. *Proc. ACM Softw. Eng.*, 1(FSE): 631–653, 2024.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *ICSE*, pp. 1469–1481. IEEE, 2023.

Michael Fu. Toward more effective deep learning-based automated software vulnerability prediction, classification, and repair. In *ICSE Companion*, pp. 208–212. IEEE, 2023.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial networks. *CoRR*, abs/1406.2661, 2014.

Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Anh Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair. *Proc. ACM Softw. Eng.*, 1(FSE):1471–1493, 2024.

Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. Inferring program transformations from singular examples via big code. In *ASE*, pp. 255–266. IEEE, 2019.

Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of API evolution literature. *ACM Comput. Surv.*, 54(8):171:1–171:36, 2022.

Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How does web service API evolution affect clients? In *ICWS*, pp. 300–307. IEEE Computer Society, 2013.

Xuan Li, Shuai Yuan, Xiaodong Gu, Yuting Chen, and Beijun Shen. Few-shot code translation via task-adapted prompt learning. *J. Syst. Softw.*, 212:112002, 2024.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pp. 74–81, 2004.

Fang Liu, Jia Li, and Li Zhang. Syntax and domain aware model for unsupervised program translation. In *ICSE*, pp. 755–767. IEEE, 2023.

Na Meng, Miryung Kim, and Kathryn S. McKinley. Sydit: creating and applying a program transformation from an example. In *SIGSOFT FSE*, pp. 440–443. ACM, 2011.

Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *ICSE*, pp. 502–511. IEEE Computer Society, 2013.

Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. In *EuroSys*, pp. 59–71. ACM, 2006.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *ICSE*, pp. 82:1–82:13. ACM, 2024.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, pp. 311–318. ACL, 2002.

Maja Popovic. chrf: character n-gram f-score for automatic MT evaluation. In *WMT@EMNLP*, pp. 392–395. The Association for Computer Linguistics, 2015.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020.

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *ICSE*, pp. 404–415. IEEE / ACM, 2017.

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. Reassessing automatic evaluation metrics for code summarization tasks. In *ESEC/SIGSOFT FSE*, pp. 1105–1116. ACM, 2021.

Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *CVPR*, pp. 815–823. IEEE Computer Society, 2015.

Lucas Serrano, Van-Anh Nguyen, Ferdian Thung, Lingxiao Jiang, David Lo, Julia Lawall, and Gilles Muller. SPINFER: inferring semantic patches for the linux kernel. In *2020 USENIX Annual Technical Conference*, pp. 235–248. USENIX Association, 2020. URL `https://www.usenix.org/conference/atc20/presentation/serrano`.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *ICSE*, pp. 25–36. IEEE / ACM, 2019.

Aäron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748, 2018.

Jiaying Wang, Lijun Cao, Jing Shan, Xiaoxu Song, and Junyi Jiang. Dual learning model of code summary and generation based on transformer. In *WISA*, volume 14883 of *Lecture Notes in Computer Science*, pp. 41–52. Springer, 2024.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. In *NeurIPS*, pp. 6559–6569, 2019.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.*, 1(FSE):1585–1608, 2024.

Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. Leveraging code generation to improve code retrieval and summarization via dual learning. In *WWW*, pp. 2309–2319. ACM / IW3C2, 2020.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *MSR*, pp. 476–486. ACM, 2018.

Imam Nur Bani Yusuf, Diyanah Binte Abdul Jamal, and Lingxiao Jiang. Automating arduino programming: From hardware setups to sample source code generation. In *MSR*, pp. 453–464. IEEE, 2023.

Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Trans. Softw. Eng. Methodol.*, 33(2):55:1–55:69, 2024.